Generator liczb losowych

Rachunek Prawdopodobieństa i Statystyka

Paweł Pyciński

Uniwersytet Jagielloński

TABLE OF CONTENTS

Cel Projektu

Celem projektu jest stworzenie generatora całkowitych liczb pseudolosowych o rozkładzie równomiernym. Na podstawie stworzonego generatora należy stworzyć generatory o rozkładzie jednostajnym (na przedziale [0,1]), Bernoulliego, dwumianowego, Poissona, wykładniczego i normalnego. Następnie należy przetestować powstałe generatory.

UNIWERSYTET JAGIELLOŃSKI W KRAKOWIE

Definicja

Generator liczb pseudolosowych – program lub podprogram, który na podstawie niewielkiej ilości informacji generuje deterministycznie ciąg bitów, który pod pewnymi względami jest nieodróżnialny od ciągu uzyskanego z prawdziwie losowego źródła.

Generator liczb pseudolosowych nie bez powodu jest **pseudolosowy**, problem z otrzymaniem liczb losowych wynika z deterministycznego charakteru komputera i wykonywanych przez niego operacji. Gdy człowiek dokonuje rzutu kością, nie wie co wypadnie. Taka sama operacja na komputerze wymaga działania, którego wynik jest nieprzewidywalny – żadna z operacji wykonywanych przez procesor nie posiada takiej cechy.

Problem starano się rozwiązać wykorzystując zewnętrzne źródła sygnałów losowych (np. generatory białego szumu), jednakże w tego typu urządzenia nie są standardowo wyposażano komputery osobiste. Próbowano także wykorzystać szumy kart dźwiękowych, jednakże system ten nie rozpowszechnił się z prostej przyczyny – różne karty dźwiękowe szumią różnie, a te z górnej półki nie szumią prawie wcale.

Sposoby generowania liczb pseudolosowych

Jest wiele sposobów generowania liczb pseudolosowych. Jedną z grup generatorów są generatory liniowe. tworzą ciąg liczb według schematu:

$$X_{n+1} = (a_1 X_n + a_2 X_{n-1} + \ldots + a_k X_{n-k+1} + c) mod(m)$$

gdzie $a_1, \ldots, a_k$, c, m -parametry generatora (ustalone liczby)

Generatory używające operacji modulo nazywamy **kongruencyjnymi**. Każdy kolejny wyraz (liczba pseudolosowa) w generatorze liniowym to suma pewnych poprzednich wyrazów pomnożonych każdy z każdą o jakiś skalar i brane z nich jest modulo.

Generator mulitplikatywny tworzy liczby według schmatu:

$$X_{i+1} = (aX_i + c) mod(m) \iff c = 0$$

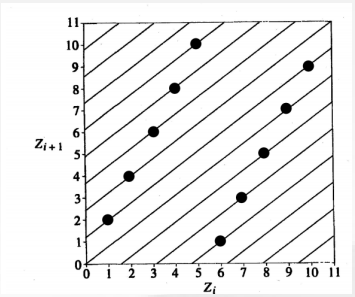
Kolejny wyraz tworzymy po przez pomnożenie poprzedniego przez jakiś skalar. Gdy $c \neq 0$ to generator jest kongurentnie mieszany.

Własny generator

Swój generator postanowiłem zbudować na bazie generatora mulitplikatywnego. Jest to jeden z łatwiejszych generatorów, prosty do implementacji.

Posiada on niestety dwie poważne wady:

1. Gererator generuje liczby ciągu w sposób deterministyczny przez co łatwo jest wyliczyć kolejną liczbę.
2. Wybierając złe czynniki możemy spowodować, że okres generatora będzie mały przez co będzie działał niepoprawnie lub będzie generował bardzo mało liczb losowych.
3. Generowane liczby lokalizują się na hiperpłaszczyznach, których położenie uzależnione jest od parametrów generatora.



Przez wyżej wymienione czynniki nie może być on stosowany w kryptografii.

Własny generator

Przed zaimplementowaniem pozostał jeszcze wybór m oraz a dla naszego generatora.

Niech $m = 2^{32}$, jest to liczba o 1 większa od zakresu unsigned int'a, dzięki czemu nasza kongruencja potencjalnie będzie mogła zwracać wszyskie liczby które jesteśmy w stanie zapisać na 4 bajtach float'a w większości języków programowania. Ponadto niech $a = 747796405$.

TABLE 5. LCGs with Good Figures of Merit, for $m = 2^e$ and $c = 0$

m	a, a^*	$M_8(m,a)$	$M_{16}(m,a)$	$M_{32}(m,a)$
2^{30}	177911525, 17372909	0.74878 *	0.53850	0.53850
	156051869, 52274357	0.69501	0.67940 *	0.64413
	143133861, 233896749	0.69305	0.66791	0.66791 *
2^{31}	594156893, 452271861	0.75913 *	0.50244	0.50244
	558177141, 413965533	0.68978	0.68749 *	0.59450
	602169653, 448899357	0.67295	0.67116	0.67116 *
2^{32}	741103597, 887987685	0.75652 *	0.53707	0.53707
	1597334677, 851723965	0.70068	0.67686 *	0.64694
	747796405, 204209821	0.66893	0.66001	0.66001 *

Własny generator - kod źródłowy

```
1     class generator:
2
3         def __init__(self, seed):
4             self.value = seed
5             self.a = 747796405
6             self.m = 4294967296
7
8         def generateRandom(self):
9             self.value = (self.a*self.value) % self.m
10            return self.value
11
```

Listing 1: Klasa generatora

Klasa generatora posiada konstruktor który jako argument przyjmuje ziarno czyli dowoloną liczbę początkową która rozpocznie budowanie pseudolosowy ciąg. Jest także metoda która zwaraca kolejną wygenerowaną liczbę.

Rozkład jednostajny

Aby uzyskać liczby z rozkładu jednostajnego na przedziale $[0, 1]$ wystarczy podzielić przez ustalone wcześniej $m = 4294967296$, zauważmy że po takiej operacji największą liczbą możliwą do uzyskania będzie 1, natomiast pozostałe liczby będą należały to przedziału $[0, 1]$.

```
1     def uniformDistribution(self):
2         return self.generateRandom()/self.m
3
```

Listing 2: Metoda rozkładu jednostajnego

Rozkład Bernoulliego

Rozkład Bernoulliego, jest rozkładem dwupunktowym, aby uzyskać ten rozkład skorzystam z metody którą przygotowałem dla rozkładu jednostajnego. Ustalmy dowolne $P \in [0, 1]$. Jeśli wylosowana liczba przez metodę rozkładu jednostajnego będzie mniejsza od p to zwrócimy 0, w przeciwnym razie 1.

```
1       def bernoulliDistribution(self, probability):
2           rand = self.uniformDistribution()
3           if ( rand<= probability):
4               return 0
5           else:
6               return 1
7
```

Listing 3: Metoda rozkładu Bernoulliego

Rozkład Dwumianowy

Rozkład dwumianowy jest to liczba sukcesów w *n* próbach Bernoulliego. W implementacji wykorzystałem wcześniej przygotowaną metodę generowania próby Bernoulliego, wywołanie jej *samples* razy daje nam rozkład Dwumianowy

```
1     def binomialDistribution(self,probablity, n):
2         counter = 0
3         for i in range(n):
4             counter+=self.bernoulliDistribution(probablity)
5         return counter
6
```

Listing 4: Metoda rozkładu Dwumianowego

Rozkład Poissona

Rozkład Poissona modeluje zdarzenia rzadkie. Jest on parameryzowany zmienną λ która jest równa oczekiwanej liczbie zdarzeń w danym przedziale czasu. Jest wiele algorytmów generujących ten rozkład na potrzeby naszego generatora wystaczy zastosować najprostszy z nich czyli **Algorytm Knutha**

```
1      def poissonDistribution(self, lambdapoiss):
2          limit = math.exp(-lambdapoiss)
3          n = 0
4          p = self.uniformDistribution()
5          while(p>=limit):
6              n+=1
7              p*=self.uniformDistribution()
8          return n
9
```

Listing 5: Metoda rozkładu Poissona

Rozkład Wykładniczy, wprowadzenie

Rozkład wykładniczy modeluje czas między kolejnymi zdarzeniami, jeśli w jednostce czasu zachodzi średnio λ niezależnych zdarzeń.

Metodę generującą rozkład wykładniczy możemy uzyskać stosując metodę odwórconej dystrybuanty. Dystrybuanta określonego rozkładu prawdopodobieństwa jest funkcją $F : \mathbb{R} \to \mathbb{R}$ niemalejąca i prawostronnie ciągła Dystrybuanta jednoznacznie definiuje rozkład prawdopodobieństwa i ma następujący związek z gęstością prawdopodobieństwa: $F(x) = \int_x^{-\infty} f(y)\,dy$. Jeśli uda się znaleźć F^{-1} to $U = F(x) \to x = F^{-1}(U)$ zmienna losowa x ma rozkład o dystrybuancie F, U jest zmienną losową o rozkładzie jednostajnym. Krótki dowód dlaczego tak jest:

Niech $X = F^{-1}(U)$ zmienna losowa

$$\begin{aligned} P\{X \leq x\} &= P\{F^{-1}(U) \leq x\} \\ &= P\{U \leq F(x)\} \\ &= F(x) \end{aligned}$$

Przejdźmy teraz do rozkładu wykładniczego. Jego gęstość prawdopodobieństwa dana jest wzorem:

$$f(x) = e^{-x}, x \in [0, \infty)$$

Natomiast dystrybuanta jest całką z funkcji gęstości.

$$F(x) = \int_{x}^{0} e^{-x}\, dx = 1 - e^{-x}$$

$$F(x) = 1 - e^{-x} = U$$

$$e^{-x} = 1 - U$$

$$F^{-1}(x) = x = -\ln(1 - U)$$

$$U \in (0, 1) \rightarrow x \in (0, \infty)$$

```
1     def exponentialDistribution(self):
2         return -math.log(1-self.uniformDistribution())
3
```

Listing 6: Metoda rozkładu Poissona

Rozkład normalny

Rozkład normalny jest jednym z najważniejszych rozkładów prawdopodobieństwa, odgrywający ważną rolę w statystyce. Przyczyną jego znaczenia jest częstość występowania w naturze. Jeśli jakaś wielkość jest sumą lub średnią bardzo wielu drobnych losowych czynników, to niezależnie od rozkładu każdego z tych czynników jej rozkład będzie zbliżony do normalnego (na podstawie CTG).

Jest wiele algorytmów aby uzyskać rozkład normalny. Ja w swojej implementacji zastosowałem polarny algorytm Boxa-Mullera nazwywany inaczej sposobem polarnym. Polega on na wylosowaniu dwóch zmiennych (x, y) z przedziału $(-1, 1)$ tak aby $0 < x^2 + y^2 < 1$ a następnie należy podstawić do wzoru:

$$x\sqrt{\frac{-2\ln s}{s}} \quad lub \quad y\sqrt{\frac{-2\ln s}{s}}$$

wzory te stosujemy na zmianę dlatego przyda się drobna modyfikacja obecnego genereatora o dodanie nowej zmiennej którą będziemy zmieniać w zależności o zastosowanego wzoru.

Rozkład normalny - kod źródłowy

```
1     def normalDistribution(self):
2     if(self.whichOne == 1):
3         self.whichOne = 0
4         return self.prevValue
5     else:
6         x = self.uniformDistribution()*2-1
7         y=self.uniformDistribution()*2-1
8         s=x*x+y*y
9         while (s>=1 or s==0):
10            x = self.uniformDistribution()*2-1
11            y=self.uniformDistribution()*2-1
12            s=x*x+y*y
13        s=math.sqrt((math.log(s)*(-2))/s)
14        self.prevValue=y*s
15        self.whichOne=1
16    return x*s
17
```

Listing 7: Metoda rozkładu normalnego

Test poprawności generatora - wprowadzenie

Sources

- http://home.agh.edu.pl/~chwiej/mn/generatory_16.pdf
- https://pl.wikipedia.org/wiki/Generator_liczb_pseudolosowych
- https://eduinf.waw.pl/inf/alg/001_search/0022.php
- https://www.ams.org/journals/mcom/1999-68-225/S0025-5718-99-00996-5/S0025-5718-99-00996-5.pdf
- http://staff.iiar.pwr.wroc.pl/grzegorz.mzyk/kmi/kmi03.pdf
- https://math.stackexchange.com/questions/785188/simple-algorithm-for-generating-poisson-distribution/785200
- https://pl.wikipedia.org/wiki/Rozk\T1\lad_wyk\T1\ladniczy
-
- https://pl.xcv.wiki/wiki/Marsaglia_polar_method